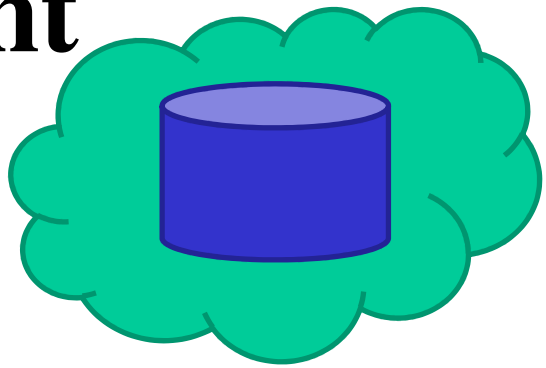
Cloud Data Management



Kapitel 6: MapReduce und Datenbanken

Dr. Michael Hartung

Sommersemester 2012

Universität Leipzig
Institut für Informatik
http://dbs.uni-leipzig.de

Inhaltsverzeichnis

SQL-Anfrageformulierung mit MapReduce

- (manuelle) Umschreibung SQL $\rightarrow$ MapReduce

- Beispiel: CouchDB
  - Dokumenten-orientierte Datenbank
  - kein Schema
  - Dokumente in JSON-Format
- Anfragen durch View-Definitionen
  - Definition von map- und reduce-Funktion in Javascript (und anderen Sprachen)

Beispieldaten

- Konzeptionell: Repräsentation als verschachtelte Tabelle

id	name	time	user	camera	info			tags
					width	height	size	
1	fish.jpg	17:46	bob	nikon	100	200	12345	[tuna, shark]
2	trees.jpg	17:57	john	canon	30	250	32091	[oak]
3	snow.png	17:56	john	canon	64	64	1253	[tahoe, powder]
4	hawaii.png	17:59	john	nikon	128	64	92834	[maui, tuna]
5	hawaii.gif	17:58	bob	canon	320	128	49287	[maui]
6	island.gif	17:43	zztop	nikon	640	480	50398	[maui]

- Intern: Repräsentation als Dokumentenmenge (JSON-Format)

```
{“_id”:”1”, "name":"fish.jpg",”time”:”17:46","user":"bob“,"camera":"nikon",
            "info":{"width":100,"height":200,"size":12345},"tags":["tuna","shark"]}
{“_id”:”2”, "name":"trees.jpg",“time”:”17:57”,"user":"john”,"camera":"canon",
            "info":{"width":30,"height":250,"size":32091},"tags":["oak"]}
....
```


Quelle: http://labs.mudynamics.com/wp-content/uploads/2009/04/icouch.html

Selektion

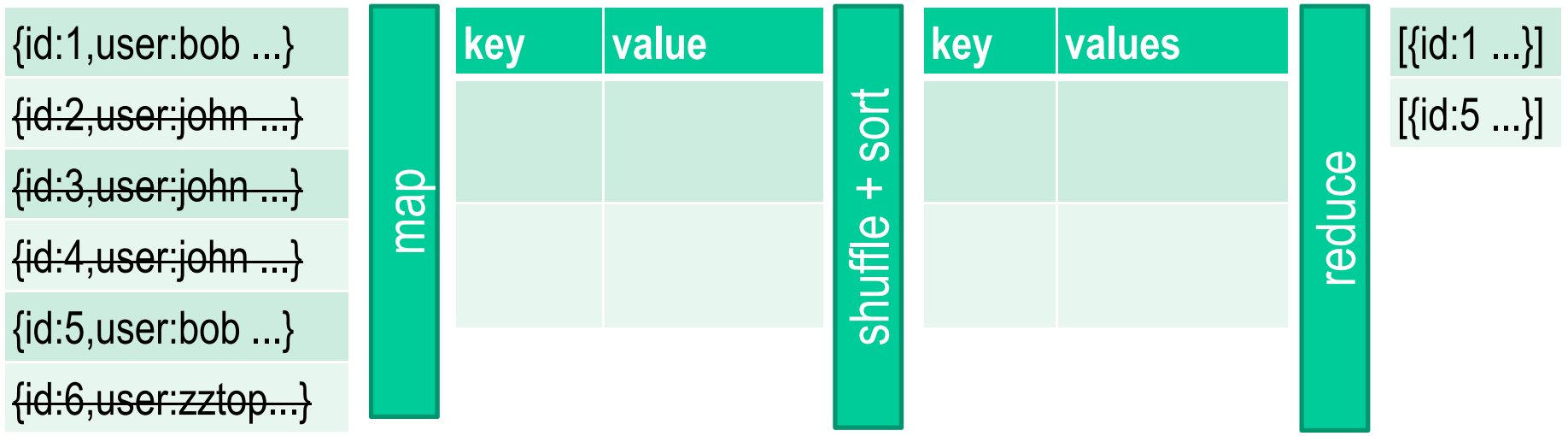
- Selektion = Bedingung für Attributwert(e)
  - SQL: ... WHERE attr = “xy”
- MapReduce
  - map: Prüfung durch IF-Bedingung, Ausgabe der selektierten Dokumente
  - reduce: Id-Funktion
- Beispiel
  - SQL: SELECT * FROM table WHERE user = “bob”

id	name	time	user	camera	info			tags
					width	height	size	
1	fish.jpg	17:46	bob	nikon	100	200	12345	[tuna, shark]
5	hawaii.gif	17:58	bob	canon	320	128	49287	[maui]

Selektion: Beispiel

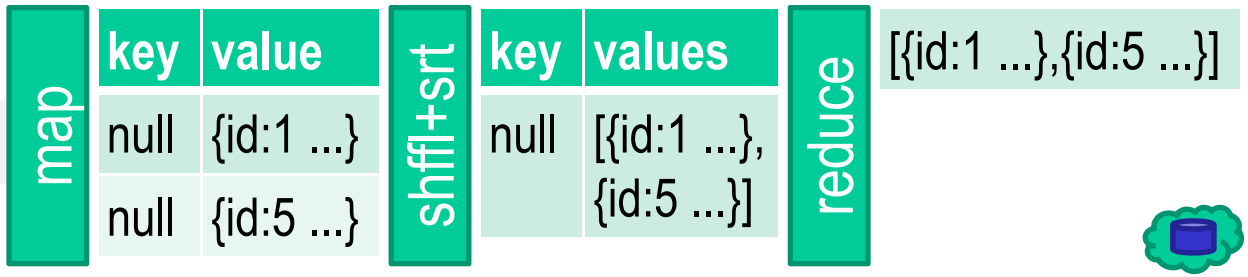
```
map
function (doc) {
  if (doc.user == “bob”)
    emit (doc.id, doc);
}
```

```
reduce
function (key, values) {
  return values;
}
```



Alternative

```
emit (null, doc);
```



Projektion

- Projektion = Einschränkung des Ergebnisses auf Attribute
  - SQL: SELECT Attr1, Attr2 FROM ...
- MapReduce
  - map: Generierung eines neuen Dokuments
  - reduce: Id-Funktion
- Duplikateliminierung
  - map: Key = Attribut(kombination)
  - reduce: Ausgabe des ersten Values
- Beispiel
  - SQL: SELECT (DISTINCT) user FROM table

user
bob
john
john
john
bob
zztop

user
bob
john
zztop

Projektion: Beispiel (ohne Duplikateliminierung)

```
map
function (doc) {
  emit(doc.id,{“user”:doc.user});
}
```

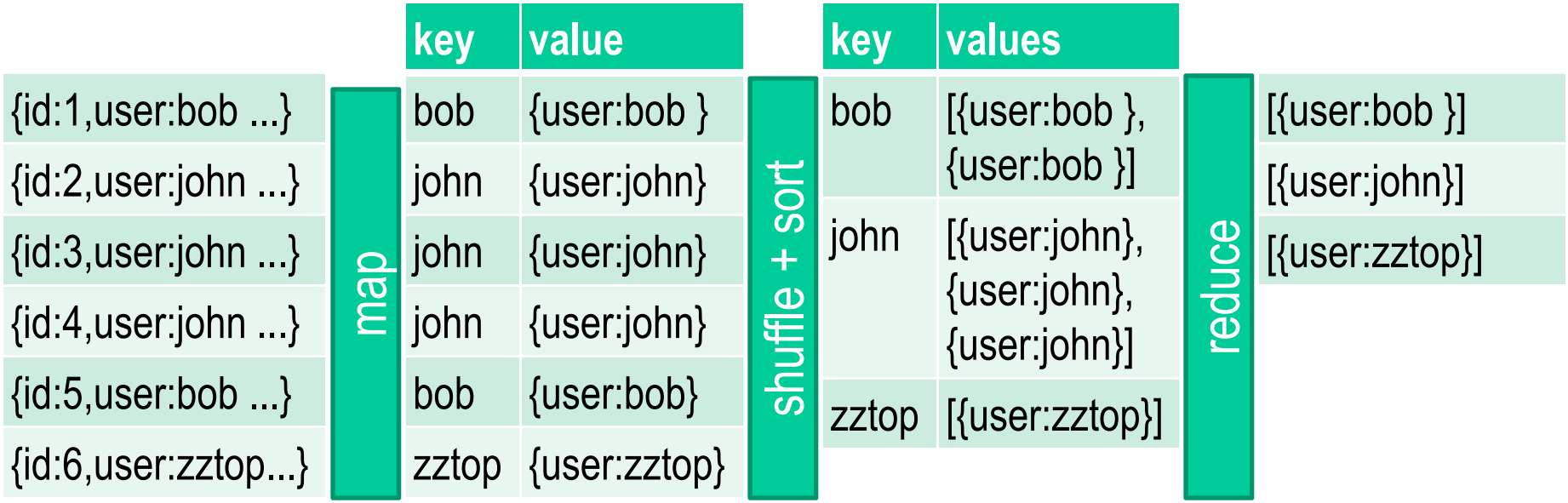
```
reduce
function (key, values) {
  return values;
}
```

		key	value		key	values		
{id:1,user:bob ...}	map	1	{user:bob }	shuffle + sort	1	[{user:bob }]	reduce	[{user:bob }]
{id:2,user:john ...}		2	{user:john}		2	[{user:john}]		[{user:john}]
{id:3,user:john ...}		3	{user:john}		3	[{user:john}]		[{user:john}]
{id:4,user:john ...}		4	{user:john}		4	[{user:john}]		[{user:john}]
{id:5,user:bob ...}		5	{user:bob}		5	[{user:bob}]		[{user:bob}]
{id:6,user:zztop...}		6	{user:zztop}		6	[{user:zztop}]		[{user:zztop}]

Projektion: Beispiel (mit Duplikateliminierung)

```
map
function (doc) {
  emit(doc.user,{“user”:doc.user});
}
```

```
reduce
function (key, values) {
  return values[0];
}
```



Gruppierung und Aggregatfunktion

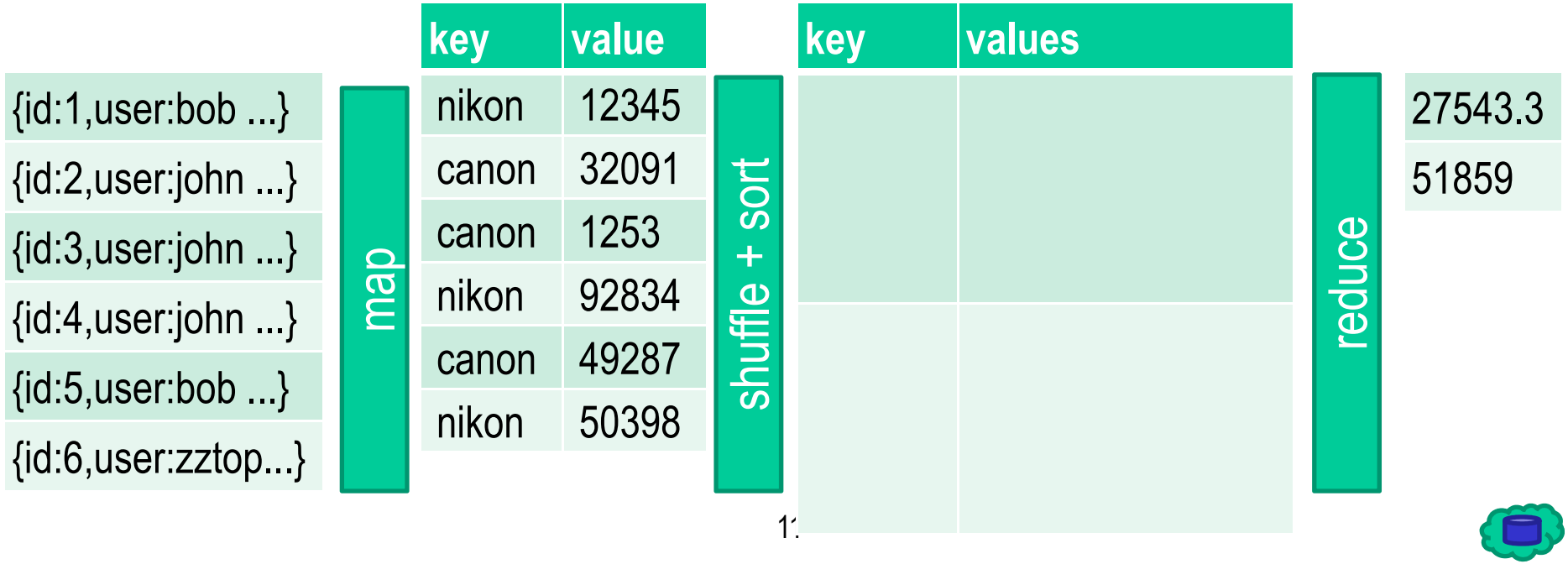
- Gruppierung
  - Zusammenfassen von Datensätzen mit gleichen Attributwerten
  - Bildung aggregierter Attributwerte pro Gruppe durch Aggregatfunktionen (z.B. SUM)
- MapReduce
  - map: Gruppierungsattribut(e) als Key
  - reduce: Anwendung der Aggregatfunktion
- Beispiel
  - SELECT camera, AVG(info.size) as avgsize
    FROM Table
    GROUP BY camera

camera	avgsize
canon	27543.3
nikon	51859

Gruppierung und Aggregatfunktion: Beispiel

```
map
function (doc) {
   emit(doc.camera,
    doc.info.size);
}
```

```
reduce
function (key, values) {
   sum = 0;
   for (i=0; i<values.length; i++) {
      sum = sum + values[i];
   }
   return sum/values.length;
}
```



Equi-Join + Mehrwertiges Attribut

- Equi-Join = Verknüpfung zweier Relationen über Attributgleichheit
  - SQL: ... WHERE Tab1.Attr1 = Tab2.Attr2
- Mehrwertige Attribute in 1NF
  - 1:N/N:M-Beziehung = weitere Relation(en), die durch Join verknüpft werden
- MapReduce
  - map: Join-Attribut als Key
  - reduce: Iteration über Paare
- Beispiel (SQL)
  - SELECT Tab1.name AS name1, Tab2.name AS name2
    FROM table AS Tab1, table AS Tab2
    WHERE Tab1.name < Tab2.name
    AND EXISTS (
      SELECT tag FROM TagTab WHERE TagTab.id=Tab1.id
      INTERSECT
      SELECT tag FROM TagTab WHERE TagTab.id=Tab2.id
)

name1	name2
fish.jpg	17:46
trees.jpg	17:57
snow.png	17:56
hawaii.png	17:59
hawaii.gif	17:58
island.gif	17:43
hawaii.png	island.gif
hawaii.gif	hawaii.png
hawaii.gif	island.gif

Equi-Join + Mehrwertiges Attribut: Beispiel (1)

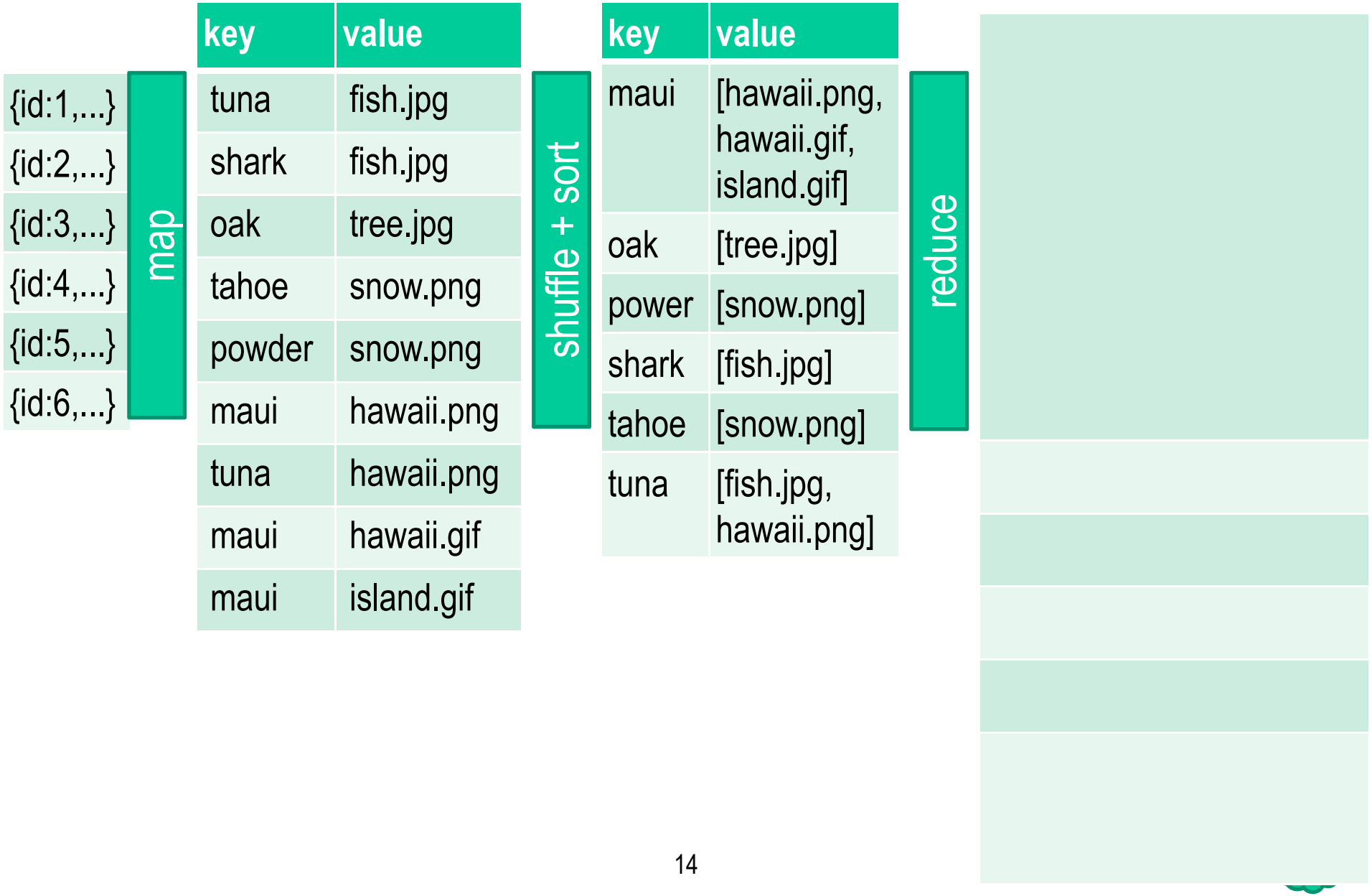
map

```
function (doc) {
  for (i=0; i<doc.tags.length; i++) {
  emit (doc.tags[i], doc.name);
}
```

reduce

```
function (key, values) {
  var result = new Array();
  for (i=0; i<values.length; i++) {
    for (k=0; k<values.length; k++) {
      if (values[i]<values[k] {
        result.push ({name1:values[i], name2:values[k]});
      }
    }
  }
  return result;
}
```


Equi-Join + Mehrwertiges Attribut: Beispiel (2)



Inhaltsverzeichnis

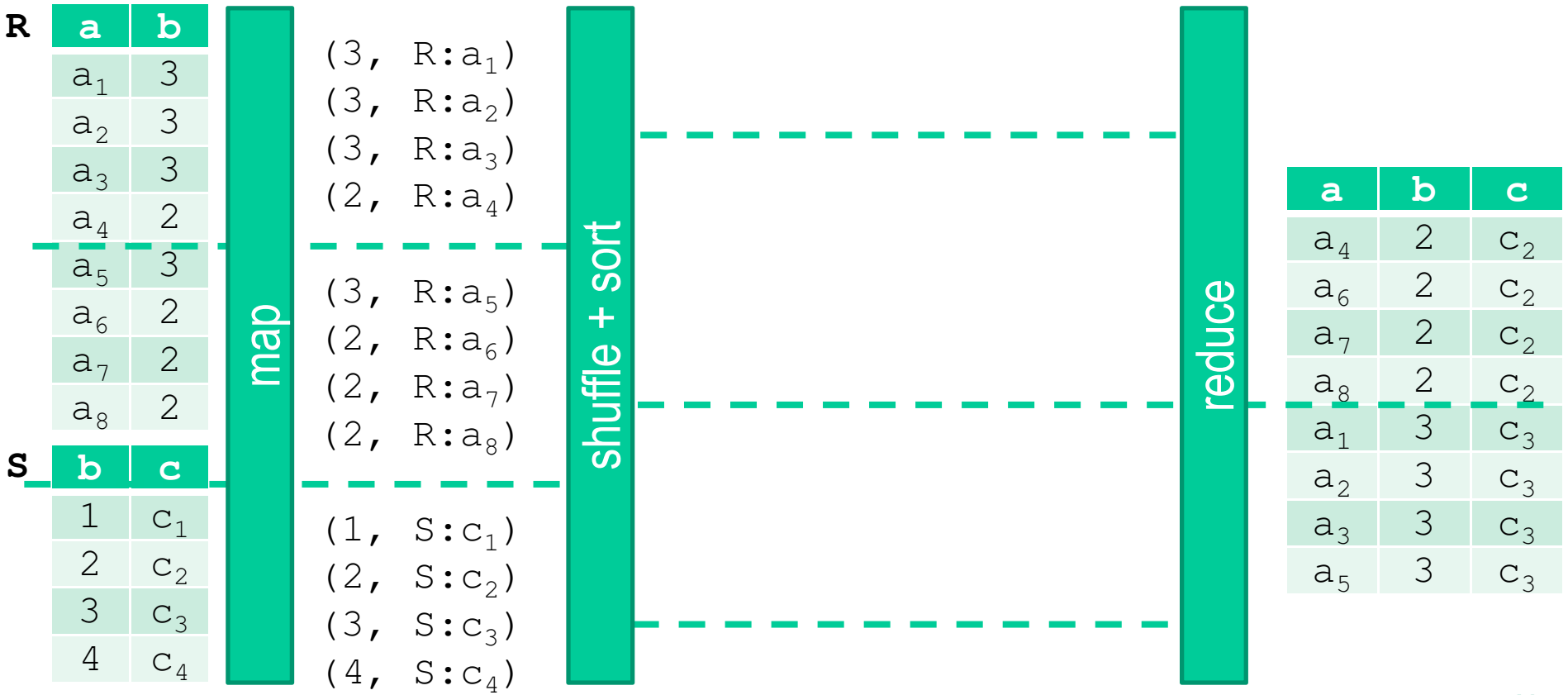
Join-Realisierung mit MapReduce

- Join ist wichtige und teure Datenbank-Operation
  - unterschiedliche Join-Arten (Natural, Outer, ...), Anzahl beteiligter Relationen
  - Fokus im Folgenden: Natural Join zwischen R und S
- Häufiger Anwendungsfall für WebApps: Logfile-Auswertung
  - Logfile ⋈ User über Attribut UserId
  - Logfile (#Klicks) meist deutlich größer als Referenztabelle (#User)
  - Geringe Join-Selektivität (nur x% der User pro Tag auf Website)
- Join-Performanz u.a. abhängig von
  - gleichmäßiger Lastbalancierung der Knoten
    → siehe Lastbalancierung bei Entity Matching
  - Datenmenge, die zwischen Map- und Reduce-Phase sortiert und transferiert wird (ggf. unter Berücksichtigung der Lokalität)
- Verschiedene Verfahren
  - Repartition Join
  - Broadcast Join
  - Semi-Join

Quelle diese und nächste Folien: [MRJoin]

Repartition Join

- Naiver Ansatz
  - map: Key=Join-Attribut, Value=Relationsname+Attribute
  - reduce: Alle Paare mit unterschiedlichem Relationsnamen



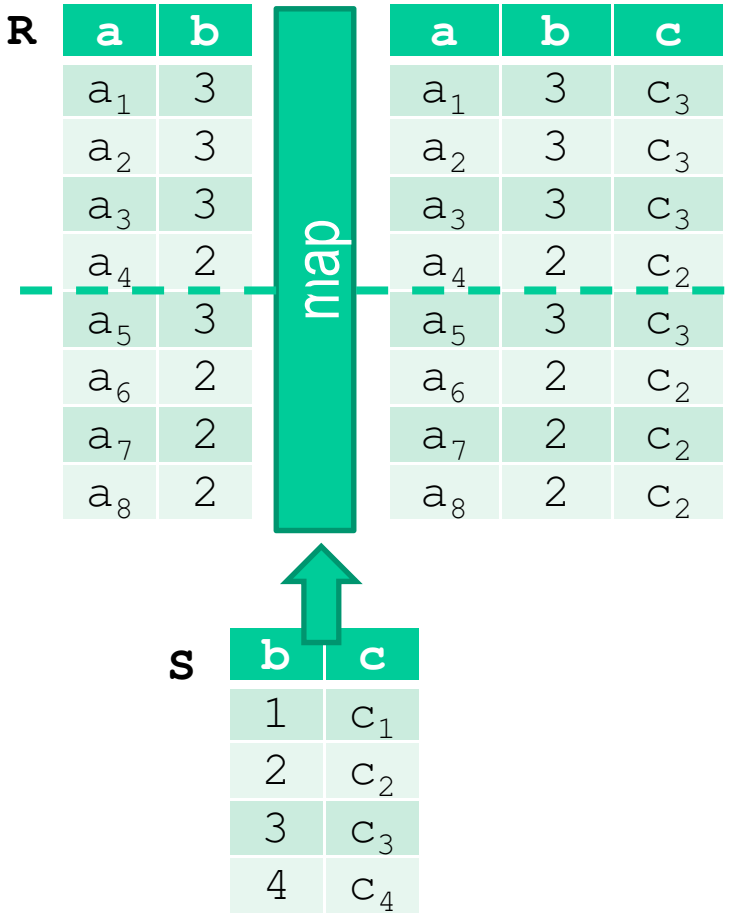
Repartition Join: Nachteile

- Alle Daten werden zwischen Map- und Reduce-Phase sortiert und an die Reduce Tasks geschickt
  - Verbesserung durch Broadcast Join, Semi-Join (nächste Folien)
- Reducer muss alle N Datensätze pro Key puffern
  - keine Reihenfolge bzgl. Input-Relation, da nur nach Join-Attribut sortiert
  - Hadoop-Implementierung erlaubt nur sequentiellen Datenzugrifff
- Lösung
  - Erweiterung des Map-Keys um Relationsname, Gruppierung nur nach Join-Attribut
  - Sortierung derart, dass Keys der kleineren Relation (*S*) vor Keys der größeren Relation (*R*) stehen → Reduce muss nur noch Datensätze von *S* puffern
- Beispiel

normal	**erweiterter Key**
(2, R:a_4)	(S:2, c_2)
(2, S:c_2)	(R:2, a_4)
(2, R:a_6)	(R:2, a_6)
(2, R:a_7)	(R:2, a_7)
(2, R:a_8)	(R:2, a_8)

Broadcast-Join

- Idee
  - Kleinere Relation (*S*) als zusätzlichen map Input
  - Join nur in map-Phase, kein Reduce notwendig
- Notwendiger Datentransfer
  - kleinere Relation muss an alle *n* Knoten geschickt werden → $n \cdot |S|$ Datensätze
  - kein Transfer von R, da jeder map-Task seine "lokale" map-Partition bearbeitet
- Vergleich Repartition-Join
  - beide Relationen werden auf alle reduce-Tasks aufgeteilt → $|R|+|S|$ Datensätze
- Dynamische Entscheidung möglich, welche Relation kleiner und ob Broadcast-Join vorteilhaft ist

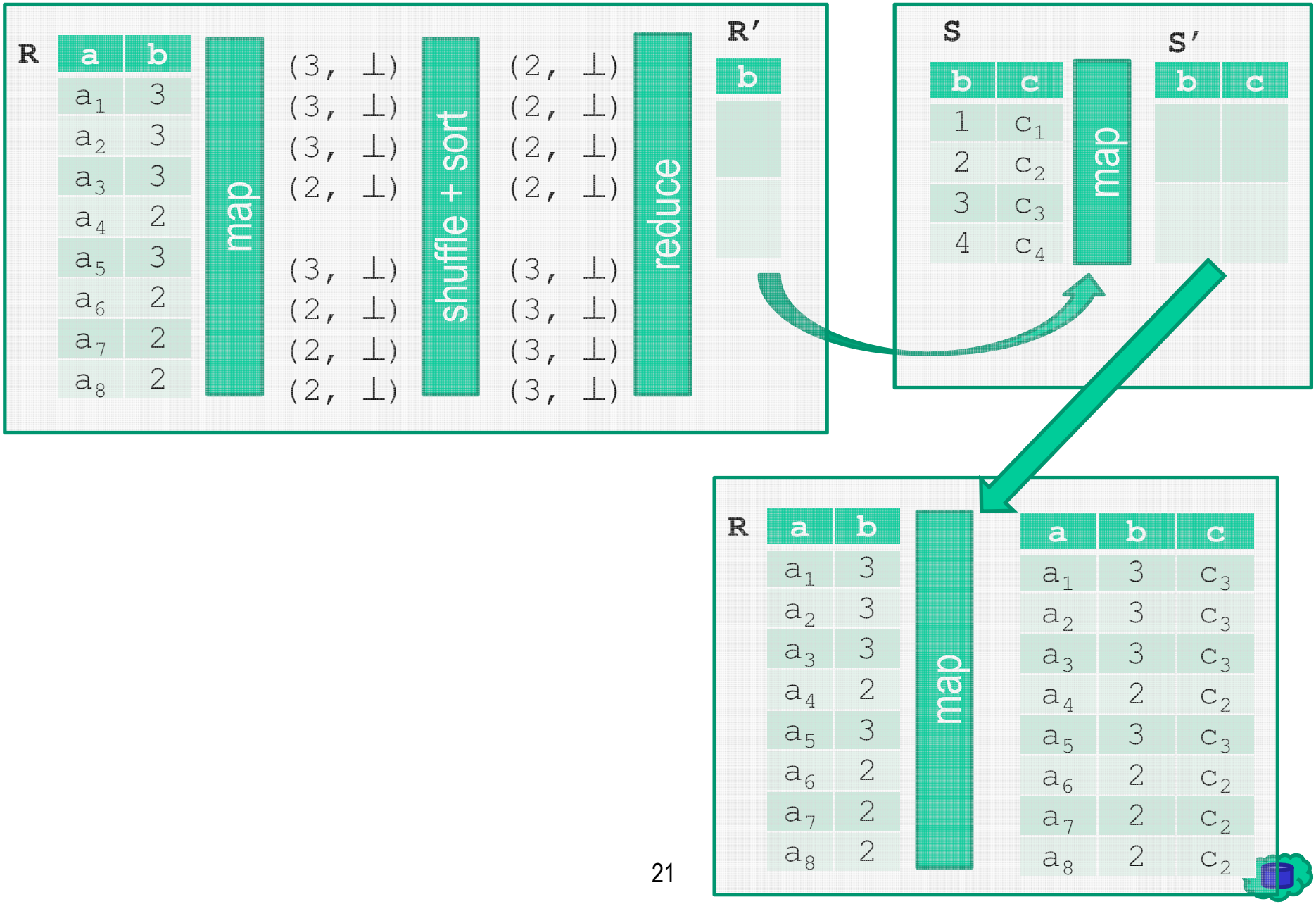


Semi-Join

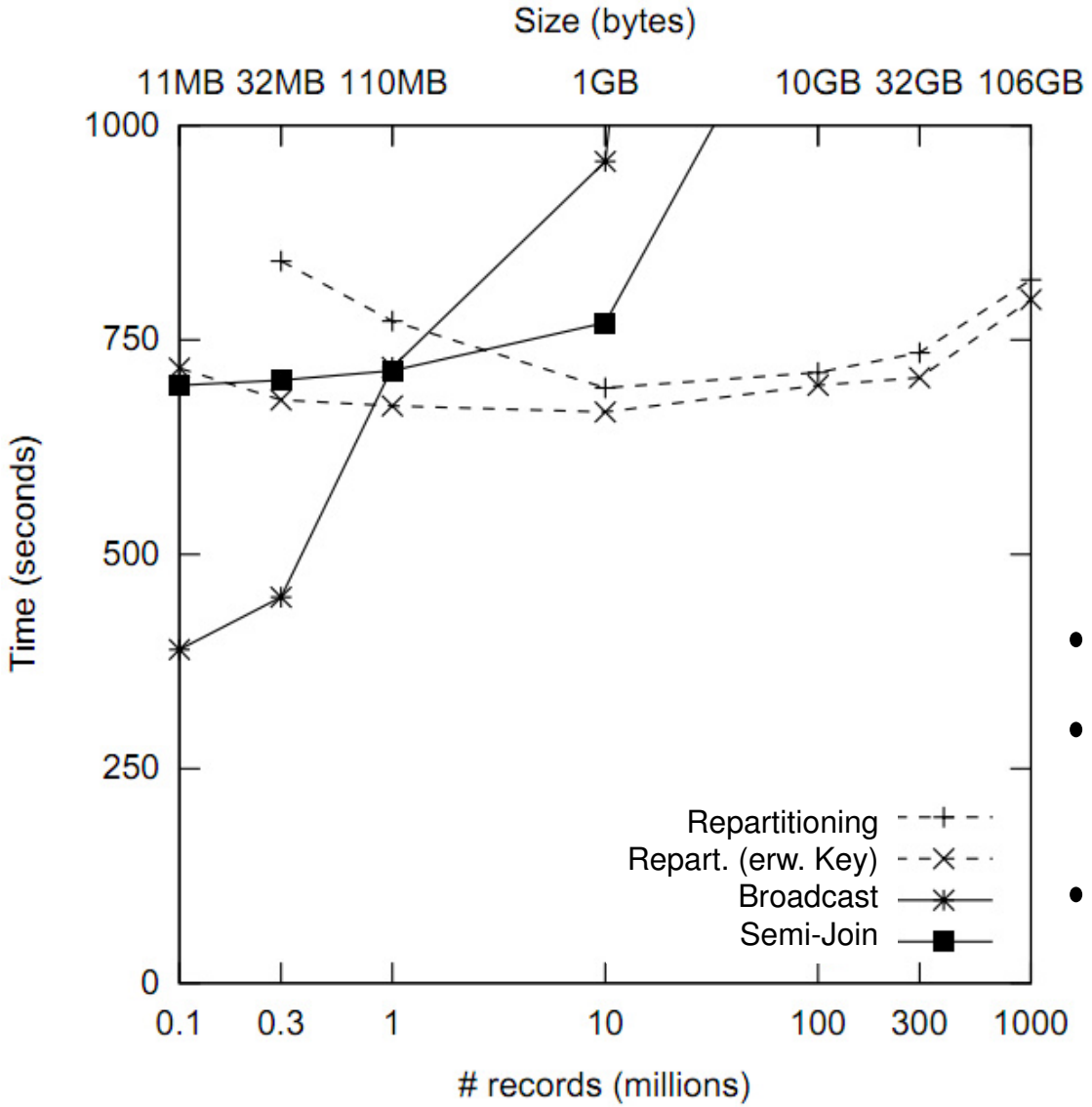
- Neben Anzahl der Datensätze auch Größe der zu transferierenden Datensätze entscheidend
  - Größe (Join-Attribut) << Größe (Datensatz)
- Semi-Join-Idee: $R \bowtie S = R \bowtie (\Pi_b (R) \bowtie S)$ mit Join-Attribut *b*
  - Realisierung durch drei Map-Reduce-Schritte

1. $R' = \Pi_b (R)$
   - map extrahiert Join-Attribut, 1 reduce task entfernt Duplikate
2. $S' = R' \bowtie S$
   - Broadcast-Join mit "kleinerer Relation" R'
3. $R \bowtie S'$:
   - Broadcast-Join mit kleinerer Relation S'

Semi-Join: Beispiel



Evaluation in [MRJoin]



#Datensätze (Relation S)	Repartition (erw. Key)	Broad-cast
0.3 Millionen	145 GB	6 GB
10 Millionen	145 GB	195 GB
300 Millionen	151 GB	6240 GB

- Datenmenge, die durch das Netzwerk geschickt wird (oben)
  - geringer als Map-Output (z.T. gleicher Knoten für map und reduce Task)

- Broadcast für kleine S
- Repartitioning: Nutzen des erweiterten Keys
- Semi-Join muss zweimal (große) Relation R einlesen

Inhaltsverzeichnis

- SQL-Anfrageformulierung mit MapReduce
- Joins mit MapReduce
- **Data Warehousing mit MapReduce**
  - Hive
- RDBMS + MapReduce
  - HadoopDB

Vergleich: Hadoop/MR vs. Parallele DBS

	Hadoop / MapReduce	Shared Nothing-RDBMS
Datengröße	PB	TB-PB
Struktur	Semistrukturierte Daten	Statisches Schema
Partitionierung	Blöcke in DFS (Byteweise)	Horizontal
Anfrage	MapReduce-Programme	Deklarativ (SQL)
Zugriff	Batch	Punkt/Bereich via Indexes
Updates	Write once read many times	Read and write many times
Scheduling		
Verarbeitung		
Datenfluss		
Fehlertoleranz		
Skalierbarkeit	Linear, unbegrenzt	Linear (existierende Setups), begrenzt
HW-Umgebung	Heterogen (preiswerte Standard-HW)	Homogen (teure High-End-Hardware)
SW-Kosten	Frei / Open Source	Sehr teuer

Vorteile: MapReduce vs. Parallele DBS

- Vorteile MapReduce
  - Skalierbarkeit/Fehlertoleranz
  - Konfigurationsaufwand
  - Kosten
  - Kein initialer Ladevorgang
- Vorteile SN-DBS
  - Deklarative Anfragesprache
  - Anfragen werden um Größenordnungen schneller beantwortet
  - (Zzt.) Arbeit auf komprimierten Daten
  - Random Access
- Typische Anwendungsfälle MapReduce
  - ETL
  - Data-Mining, Data-Clustering
  - Analyse semistrukturierter Daten (Web-Logs, …)
  - Einmal-Analysen eines Datenbestandes

Datenanalyse: Beispiel Facebook

- Facebook
  - 4TB komprimierte Daten pro Tag
  - 135TB komprimierte Daten werden pro Tag analysiert
- Aggregationen
  - Anzahl Clicks/Pageviews pro Tag/Monat/...
- Ad-hoc-Analyse
  - Wieviele Foto-Uploads zu Neujahr in den USA pro County/State?
- Data Mining
  - Nutzerverhalten als Funktion von Attributen (#Pageviews, #Sessions, Zeit, ...)
- Spam-Erkennung
  - (Verdächtige) häufige Muster in UGC (user generated content)
- Auswertung / Optimierung von Werbung
  - Anzahl AdClicks pro Nutzertyp/...

Hive

- Datenbank / Data Warehouse basierend auf Hadoop
- Hive = MapReduce + SQL
  - SQL ist einfach und weit verbreitet
  - MapReduce skaliert sehr gut
- Automatische Übersetzung von SQL nach MapReduce nötig
  - Programme schwer zu warten, kaum Reuse
  - Barriere für Nicht-Experten
  - Fehlende Ausdrucksmächtigkeit, z.B. hoher Zeitaufwand, um simple Count/Avg-Anfragen in MapReduce zu realisieren

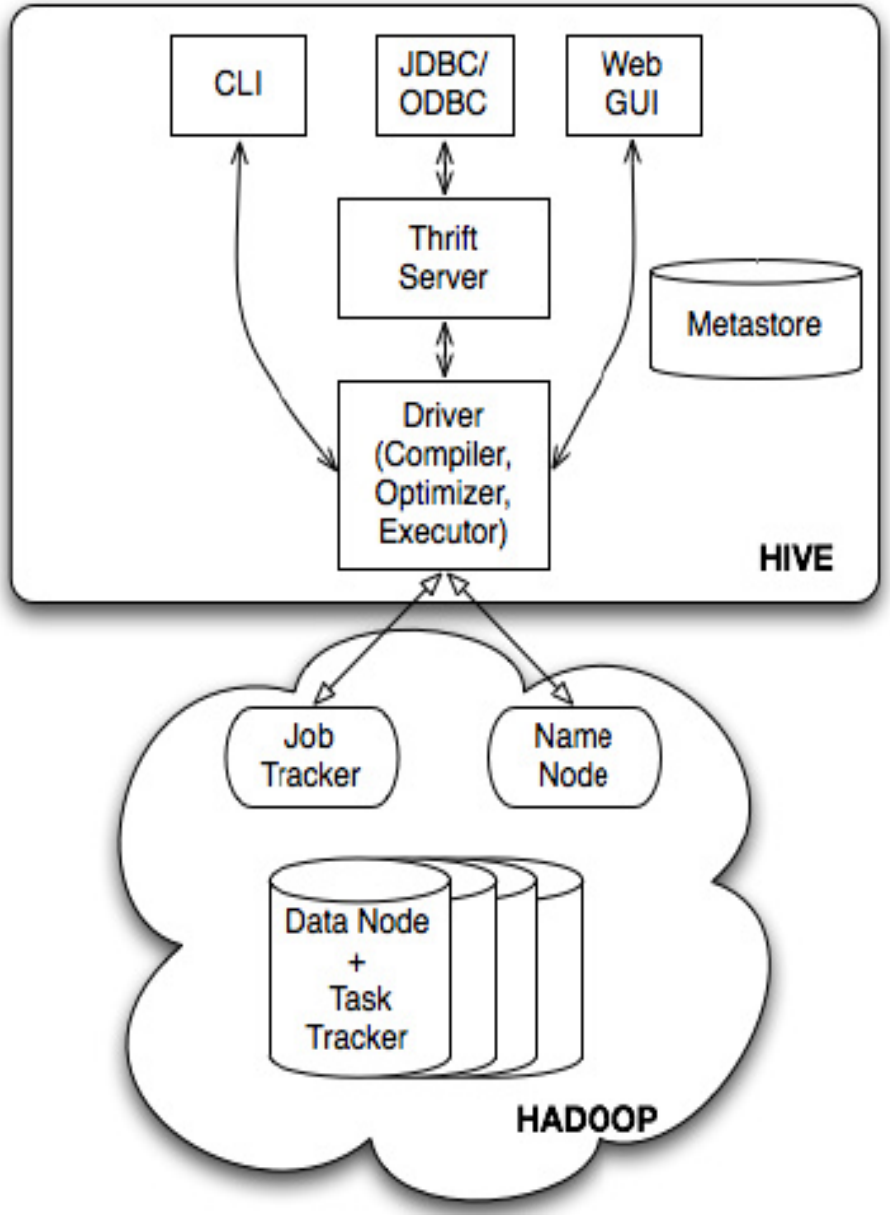
Quellen für diese und folgende Folien: [Hive], [Hive1], [Hive2], [Hive3]

Hive: Übersicht

- Verwaltung und Analyse strukturierter Daten mit Hilfe von Hadoop
  - keine Online-Datenbank, hohe Latenzzeit
- Datenhaltung im HDFS, Metadaten für Abbildung auf Tabellen
  - Komplexe Datentypen (u.a. Listen, Maps)
  - Direkter Zugriff auf Dateien und unterschiedliche Datenformate
- Anfragen mit HiveQL, Ausführung mit MapReduce
  - Einbindung von Skripten (z.B. Python) in Anfragen
  - Metadaten u.a. für Optimierung (u.a. Join, Group By)
- Skalierbarkeit und Fehlertoleranz
  - durch HDFS + MapReduce
- Erweiterbarkeit
  - User-Defined Table-Generating Functions (UDTF)
  - User-Defined Aggregate Functions (UDAF)

Hive: Architektur

- Metastore
  - Tabellen, Spalten/Typ
  - Location, Partitionen
  - (De)Serialisierungsinformationen
- CLI / Web-GUI
  - Browse Metastore
  - Absetzen von Abfragen
- Thrift
  - Cross-language Service → HiveQL
- Compiler + Optimizer
  - Anfrageoptimierung und Übersetzung des HiveQL-Statements in DAG von MapReduce Jobs
- Executor
  - Ausführung der MR-Jobs entsprechend DAG



Hive: Datentypen & Datenzugriff

- Datentypen
  - einfache und zusammengesetzte Datentypen
  - Listen, Maps
- Flexible (De)Serialisierung von Tabellen
  - unterschiedliche (nutzerdefinierte) Formate, z.B. XML, JSON, CSV
  - unterschiedliche Speicherung, z.B. Datei, ProtocolBuffer (geplant)
- Vorteil
  - keine Konvertierung (und Replizierung!) der Originaldaten in relationale Form sondern direkter Hive-Zugriff
- Nachteil
  - keine Vorverarbeitung (z.B. Indexierung) möglich
  - immer full Table/File Scans nötig

Hive: Tabellen, Partitionen und Dateien

- Tabelle kann auf exist. Daten im HDFS verweisen
  - Tabelle hat korrespond. HDFS-Verzeichnis : `/wh/pvs`
  - Definition von Spalten, anhand denen Daten partitioniert werden

    `/wh/pvs/ds=20090801/ctry=US`

    `/wh/pvs/ds=20090801/ctry=CA`
  - Bucketing: Aufteilen der Dateien eines Verz. anhand Hash-Wert (Datenparallelität)

    `/wh/pvs/ds=20090801/ctry=US/part-00000` ...

    `/wh/pvs/ds=20090801/ctry=US/part-00020`

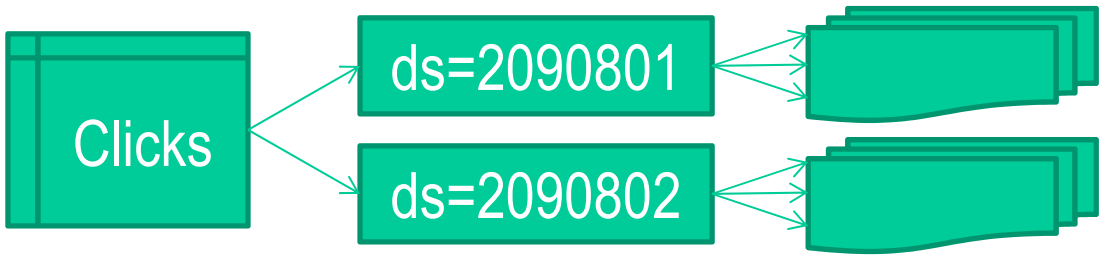


Tabelle | Partitionen (evtl. mehrere Level) | HDFS Dateien (evtl. als Hash-Bucket)

Hive: Tabellen

- Erstellung

```
CREATE EXTERNAL TABLE pvs
(userid int, pageid int, ds string, stry string)
PARTITIONED ON(ds string, ctry string)
STORED AS textfile
LOCATION ‘/path/to/existing/file’
```

- Laden von Daten

```
status_updates
(user_id int, status string, ds string)
LOAD DATA LOCAL
INPATH ‘/logs/status_updates’
INTO TABLE status_updates
PARTITION (ds=’2009-03-20’)
```


Hive-QL

- SQL-ähnliche Anfragesprache
  - Selektion, Projektion, Equi-Join, Union, Sub-Queries, Group By, Aggregatfunktionen
- Erweiterung von Queries um
  - MapReduce Skripte
  - UDF, auch auf komplexen Objekten (Lists, Map)

```
FROM (
    FROM pv_users
    SELECT TRANSFORM(pv_users.userid, pv_users.date)
    USING 'map_script'
    AS(dt, uid)
    CLUSTER BY(dt)
) map
INSERT INTO TABLE pv_users_reduced
SELECT TRANSFORM(map.dt, map.uid)
USING 'reduce_script'
AS (date, count);
```

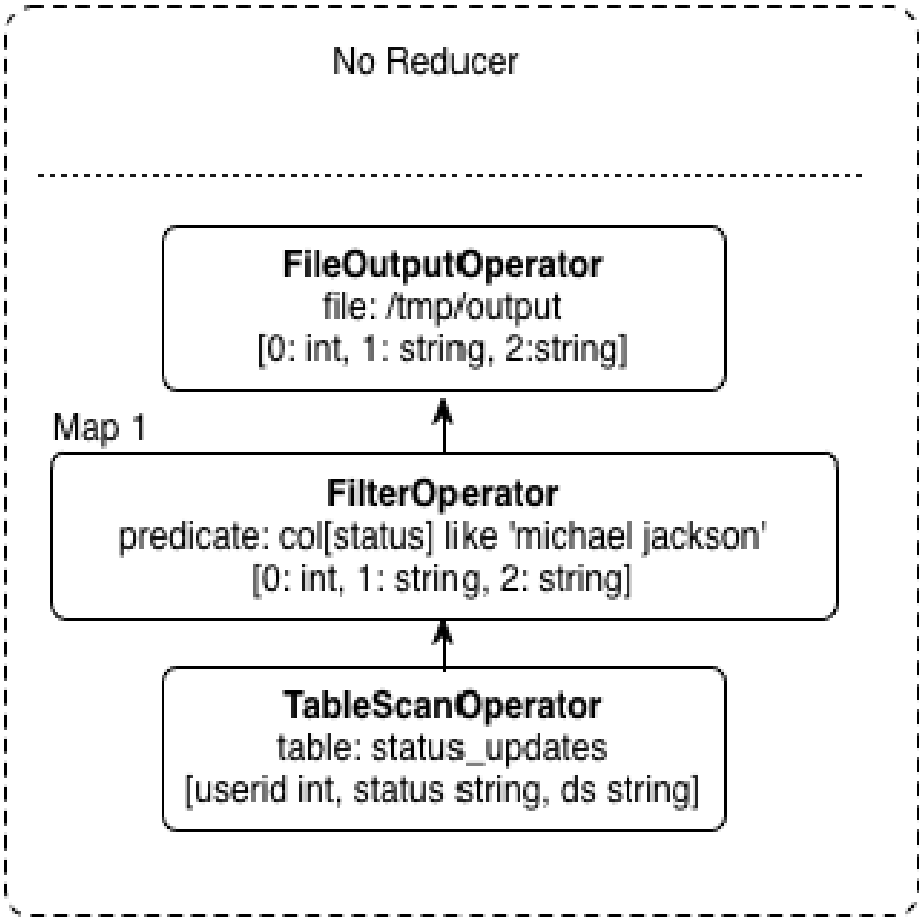

Hive-QL: Anfrageübersetzung

- Hive-QL Query wird in DAG (directed acyclic graph) übersetzt
- Knoten: Operatoren
  - TableScan
  - Select, Extract
  - Filter
  - Join, MapJoin, Sorted Merge Map Join
  - GroupBy, Limit
  - Union, Collect
  - FileSink, HashTableSink, ReduceSink
  - UDTF
- Graph repräsentiert Datenfluss
- Mehrere (parallele) Map/Reduce Phasen möglich

Hive-QL: Anfrageübersetzung (Beispiel)

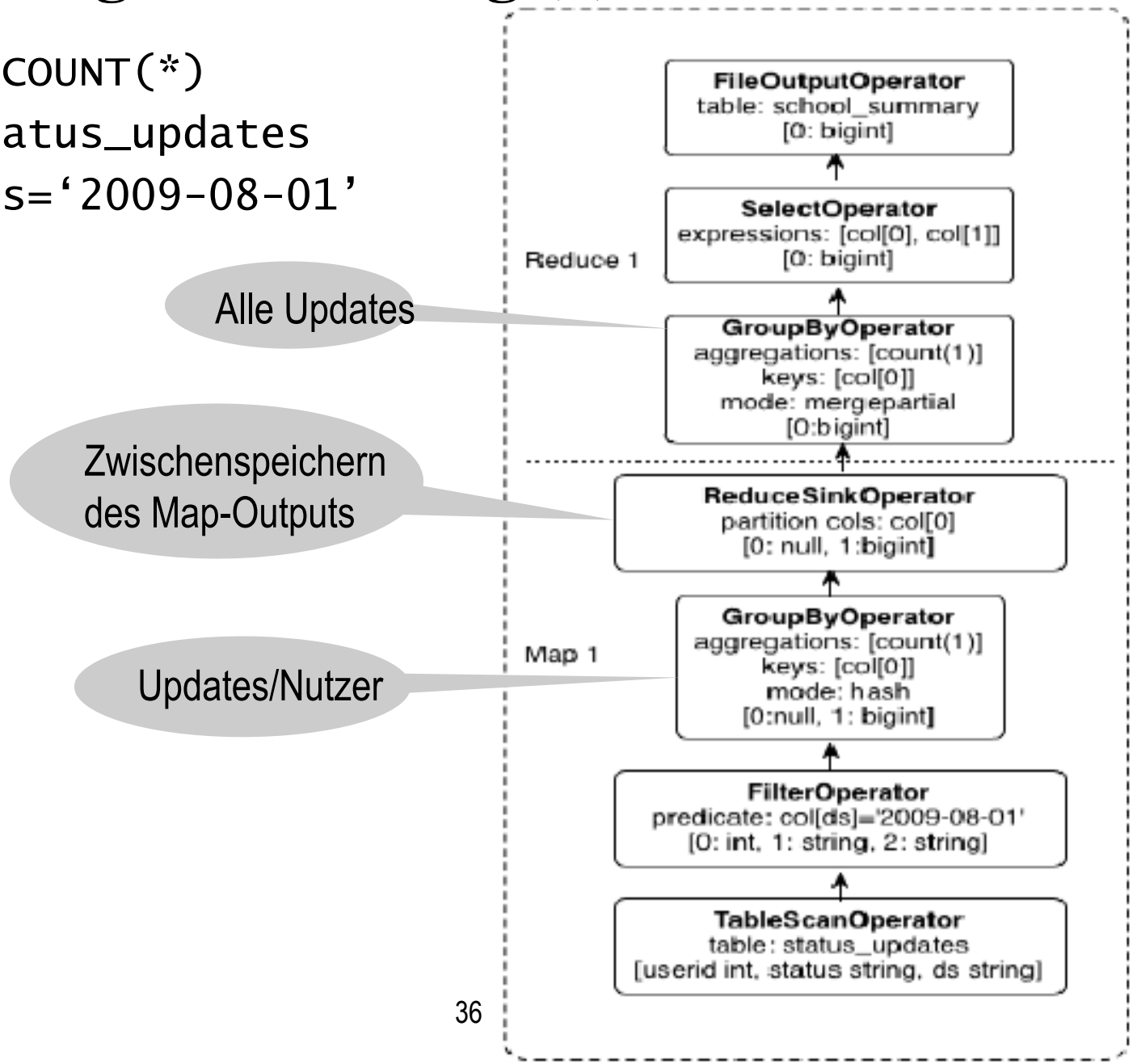
- Beispiel

```
SELECT *
FROM status_updates
WHERE status
   LIKE ‘michael jackson’
```



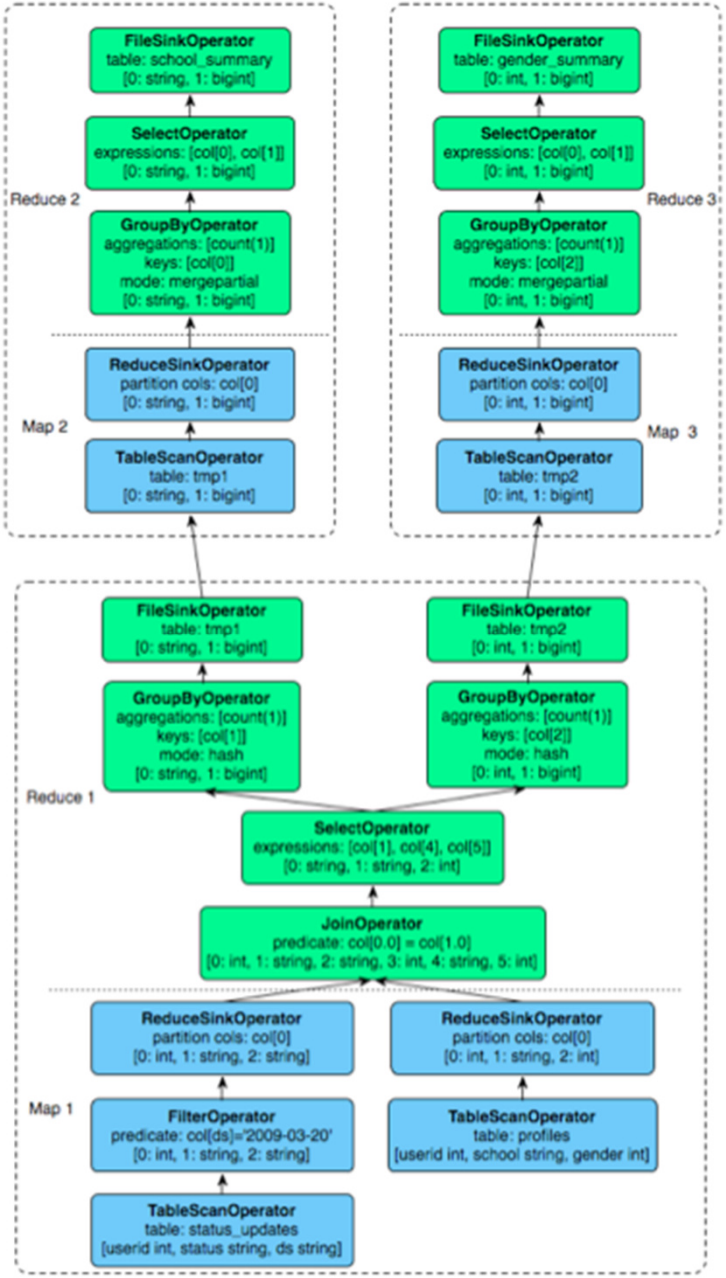
Hive: Anfrageübersetzung (2)

```
SELECT COUNT(*)
FROM status_updates
WHERE ds='2009-08-01'
```



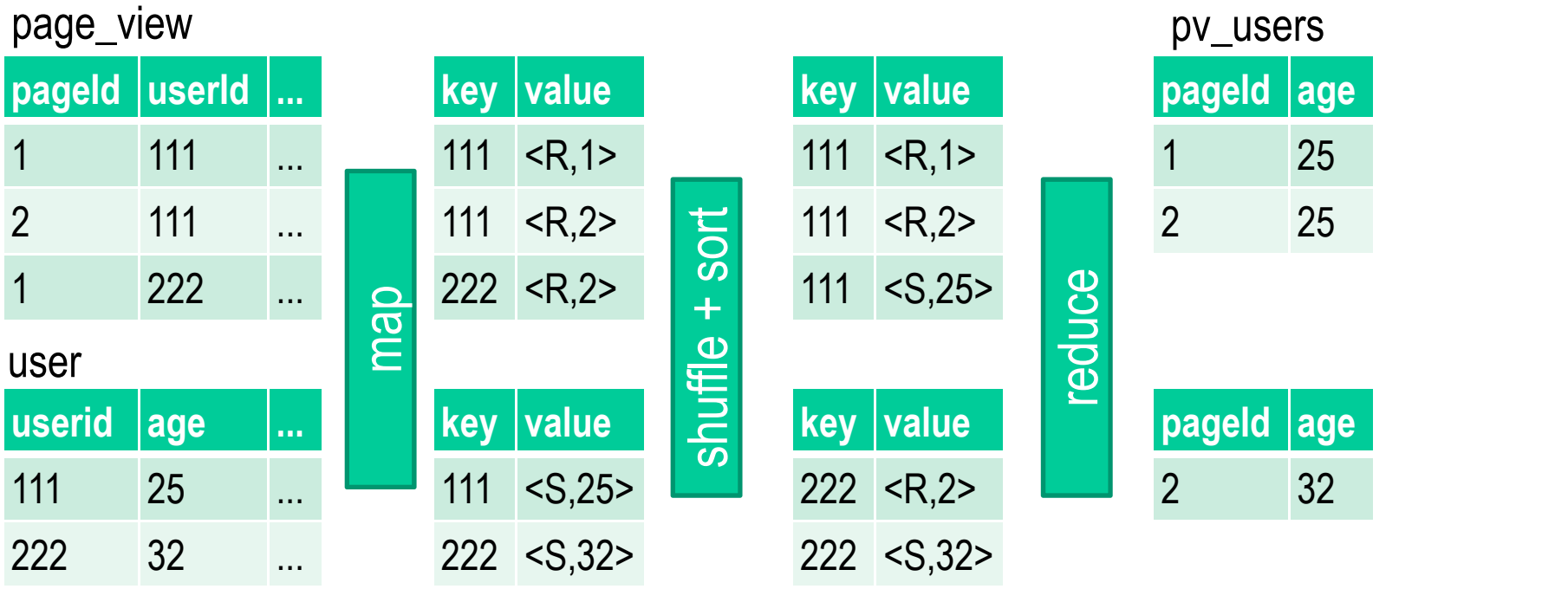
Hive: Anfrageübersetzung und -optimierung

- Anfragepläne können sehr komplex werden
- Anfrageoptimierung
  - Verwerfen nicht benötigter Spalten
    - Berücksichtigung von (Outer-)Join- und Selektionsattributen
  - Frühes Anwenden von Selektionsprädikaten
  - Verwerfen nicht benötigter Partitionen



Hive: Join

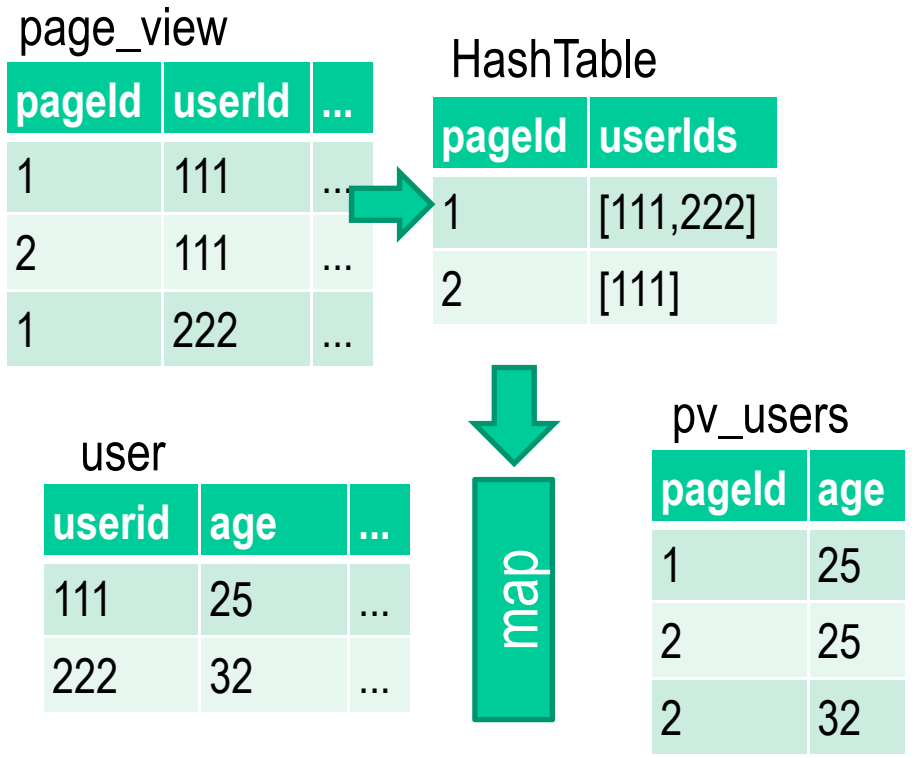
```
INSERT INTO TABLE pv_users
SELECT pv.pageid, u.age
FROM page_view pv
JOIN user u ON (pv.userid = u.userid)
```



- Key = Join-Key, Value mit Flag (R oder S) zur Unterscheidung d. Tabellen
- Mehrweg-Join mit selben Join-Key → 1 MapReduce job
- Mehrweg-Join mit *n* Join-Keys → *n* MapReduce job

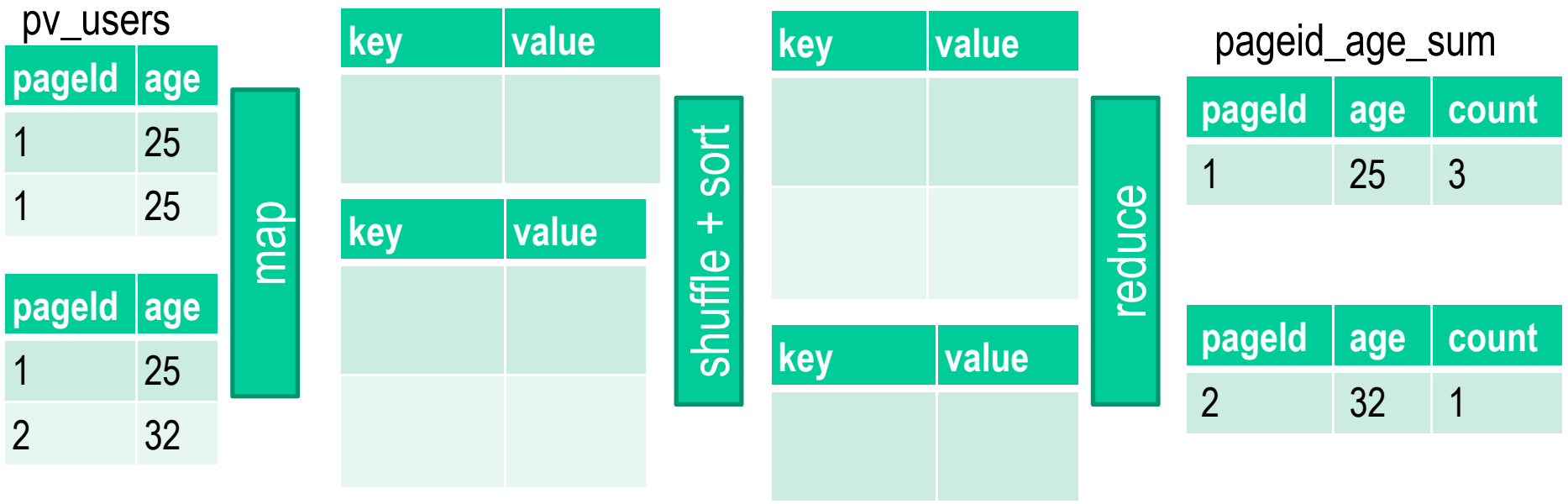
Join: Performanzsteigerung durch MapJoin

- MapJoin (aka Broadcast Join)
  - kleine Tabelle als zusätzlicher Map-Input
  - kann vorher zu Hash-Tabelle umgewandelt werden (ggf. zusätzlich komprimiert)
  - kein Reduce notwendig
  - *n* Wege-Join möglich, wenn *n*-1 Tabellen für map verfügbar
- Dynamische Join-Entscheidung
  - Bestimmung großer/kleiner Tabelle zur Laufzeit
  - Anwendung von MapJoin falls kleine Tabelle(n) “klein genug”



Hive: Group By

```
INSERT INTO TABLE pageid_age_sum
SELECT pageid, age, count(*)
FROM pv_users
GROUP BY pageid, age
```



- Key = Gruppierungsattribute
- Reduce = Aggregationsfunktion
  - “Voraggregation” durch Combiner in Map möglich (z.B (<1,25>,2))

Nutzer-definierte Skripte

- Verwendung von Skripten in HiveQL-Anfragen mittels TRANSFORM-Operator
  - Daten(de-)serialisierung
  - Austausch per stdin/stdout

id	title
1	Body Snatcher
2	Armageddon
3	AI

firstl	n
B	1
A	2

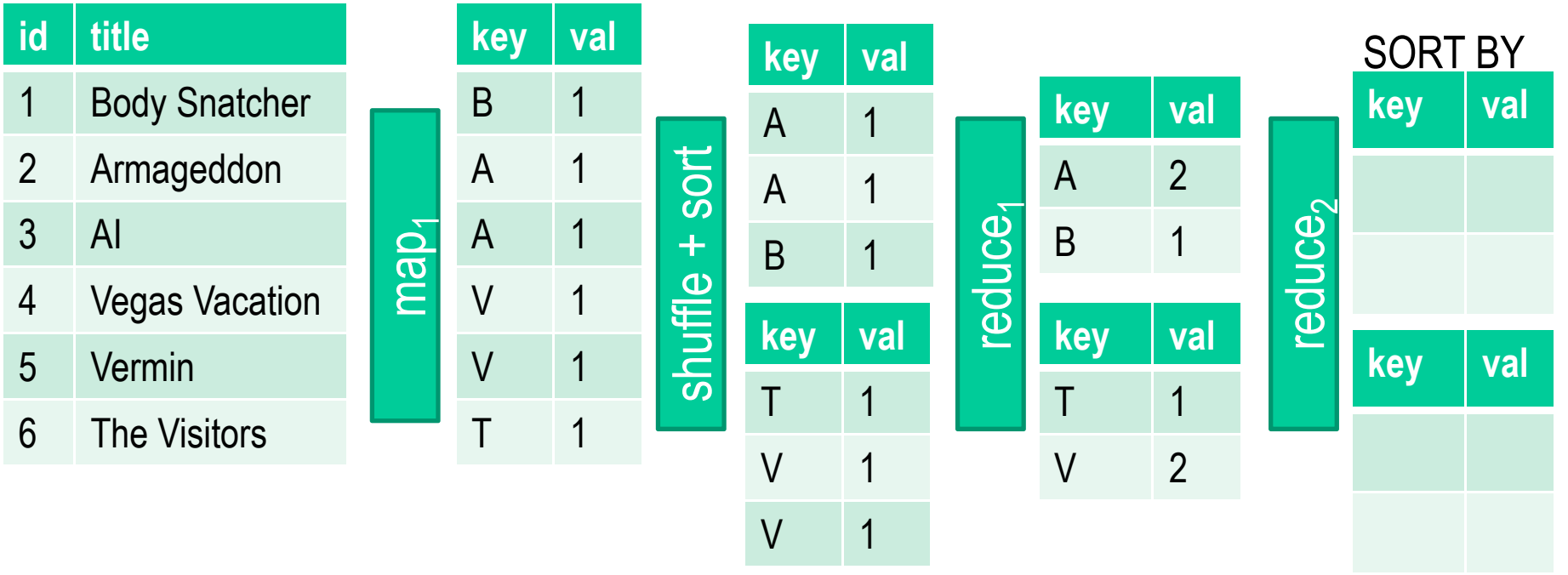
firstletter.py

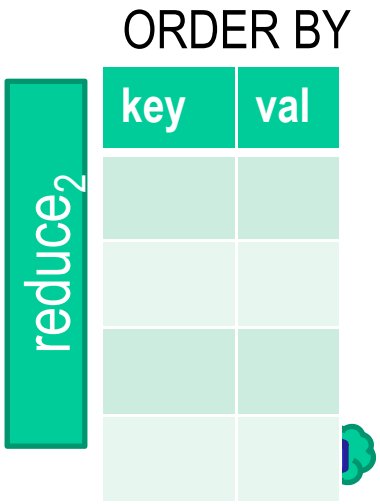
```
import sys
for line in sys.stdin:
  line = line.strip()
  id, title = line.split('\t')
  firstl = title[:1]
  print '\t'.join([id, title, firstl])
```

```
ADD FILE firstletter.py;
SELECT firstl, count(id) AS n
FROM (
  SELECT
    TRANSFORM (id, title)
    USING 'python firstletter.py'
    AS id, title, firstl
  FROM item ) f
GROUP BY firstl;
```


Hive: Sortierung

```
SELECT firstl, count(id) AS n
FROM ... GROUP BY firstl
SORT BY | ORDER BY n DESC
```





- Sortierung durch zusätzlichen Reduce-Step
- SORT BY = Sortierung pro Reducer
- ORDER BY = globale Sortierung
  = SORT BY mit nur einem Reducer

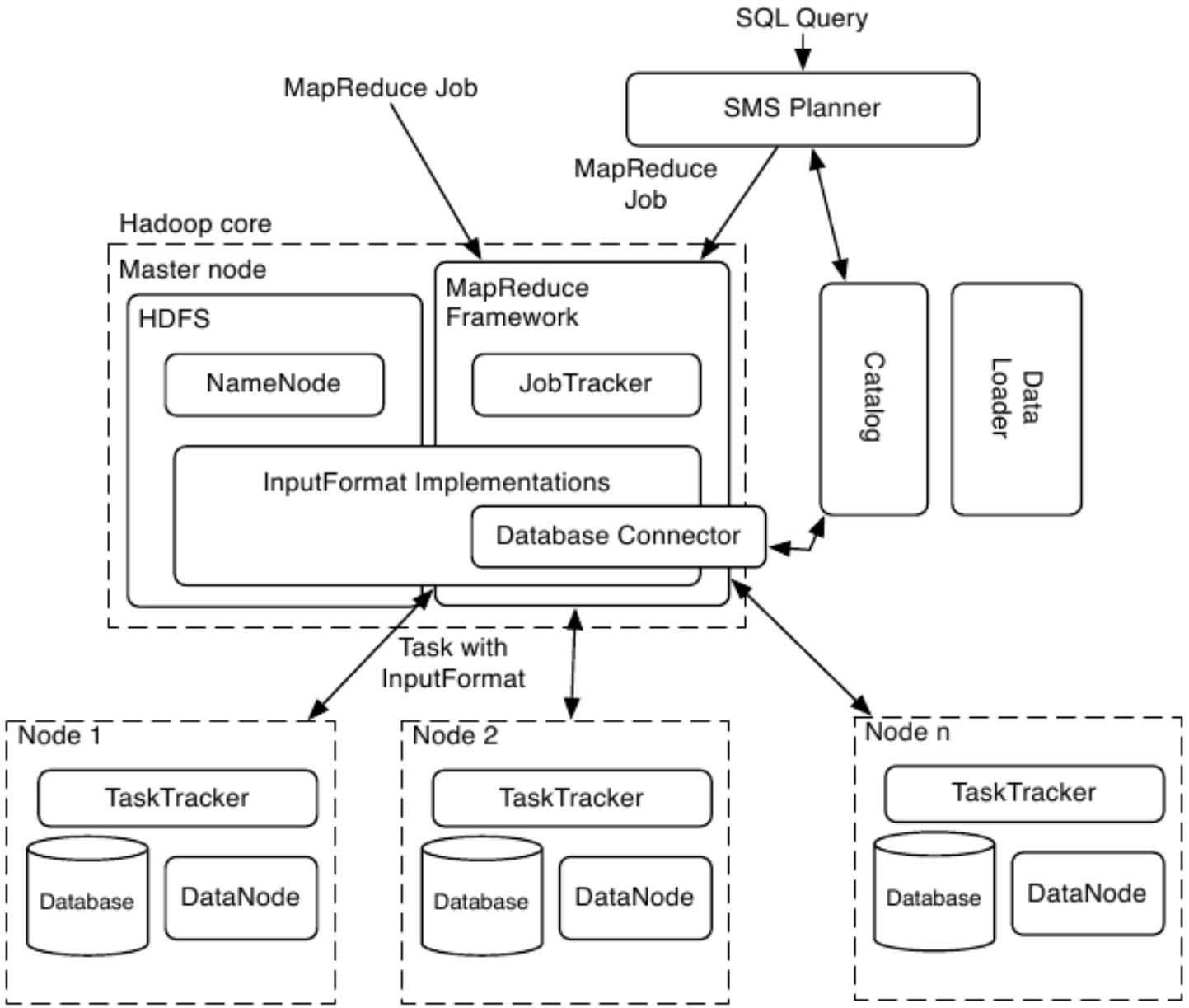
Inhaltsverzeichnis

HadoopDB

	MapReduce	Shared Nothing-RDBMS
Fehlertoleranz	Neustart des Map-/Reduce-Tasks	Query-Restart (z. T. Operator-Restart)
Skalierbarkeit	+	o
Performanz	o (u.a. TableScan auf Daten, Join, synchrone MR-Phasen, materialisierte Zwischenergebnisse, ...)	+ (u.a. effiziente Zugriff mit Indizes, asynchrones Pipelining, kostenbasierte Anfrageoptimierung, ...)

- Ziel: Kombination Fehlertoleranz+Skalierbarkeit von Hadoop mit Performance paralleler DBS
- Idee von HadoopDB
  - Viele unabhängige Single-Node DBS (PostgreSQL/MySQL)
  - Hadoop als Koordinator und Kommunikations-Layer
  - Anfragen mit MapReduce parallelisiert
  - Großteil der Arbeit wird in DBS-Nodes verrichtet

HadoopDB: Architektur



HadoopDB: Architektur (2)

- DB-Connector
  - Hadoop InputFormat-Implementierung
  - “Datenbanken für Hadoop wie HDFS-Blöcke”
- Katalog
  - Metadaten über DB (Location, Driver, …)
  - Metadaten über gehaltene Daten (Partitionierung, Replikation, …)
- Data Loader
  - Partitionierung der Daten während des Ladens
- SQL to MapReduce to SQL Planner
  - Erweitert Hive
  - Hive ist regelbasiert, ohne kostenbasierte Anfrageoptimierung
  - Anpassung der Hive Ausführungspläne
    - Erstellen von Single-Node-Queries (Optimierung durch DBS)
    - Kombination mit MapReduce

Zusammenfassung

- MapReduce ist kein DBMS, kann aber zur “datenbank-artigen” Verarbeitung großer Datenmengen genutzt werden
  - SQL-Anfragen können automatisch in MapReduce-Programme transformiert werden
  - MR kann flexibel auf die (semi-strukturierten) Originaldaten (d.h. Dateien) zugreifen
- RDBMS sind “pro Knoten” effizienter als MapReduce
  - ... aber MapReduce skaliert deutlich besser und ist fehlertoleranter
- Kombination der Stärken von RDBMS und MapReduce sinnvoll
  - ... und Gegenstand aktueller Forschung

Quellen & Literatur

- [MRJoin] Blanas et al.: A Comparison of Join Algorithms for Log Processing in MapReduce. SIGMOD 2010
- [Hive] http://hadoop.apache.org/hive/
- [Hive1] http://www.slideshare.net/zshao/hive-data-warehousing-analytics-on-hadoop-presentation
- [Hive2] http://www.slideshare.net/ragho/hive-user-meeting-august-2009-facebook
- [Hive3] http://www.slideshare.net/jsichi/hive-evolution-apachecon-2010